Research Report Number 71 - 1

Department of Computer Science
YALE UNIVERSITY

New Haven, Connecticut 06520

January, 1971

SYNTAX GRAPHS AND FAST CONTEXT FREE PARSING

by

Edgar T. Irons

SYNTAX GRAPHS AND FAST CONTEXT FREE PARSING

Edgar T. Irons

Yale University, New Haven, Connecticut

Key Words: Syntax, graph, parsing, context-free

Abstract: A graphical representation for productions of a context-free grammar is defined and used to analyze some popular parsing methods. An algorithm is presented which compares well with operator-precedence parsing programs for grammars which do not allow high degrees of local ambiguity.

SYNTAX GRAPHS AND FAST CONTEXT FREE PARSING

1. INTRODUCTION

Syntax graphs have been used both as a means for presentation of context-free grammars [1] and for storing such grammars in a computer memory for use with parsing programs [3]. In Section 2, we define a graph form for context-free grammars which is suitable for both purposes. The graphs are directly useable by a bottom-up parsing program, and can be modified in a mechanical way to be suitable for a top-down parsing program. In Section 3, we discuss the relative merits of top-down, bottom-up, and bounded context parsing programs in terms of process trees for parsing. In Section 4, we continue the comparative analysis at the more detailed level of the work required to derive nodes of a process tree. In Section 5, we describe a "fast parsing" program arising from these analyses. This program is a bottom-up multiple-tracking program which uses the graph and auxiliary tables of modest dimension to achieve a parsing speed which closely approaches that of parsing programs for bounded-context grammars. We draw heavily on the ideas developed in the report "Multiple-track Programming" [5], and recommend it as a preliminary reading.

2. THE SYNTAX GRAPH

The Syntax Graph is a tying together of the productions of a context-free grammar ([9], [4]) to allow the recognition process for sentences in the grammar to be expressed in terms of simple motions in the graph. The state of the parsing process can be completely characterized by a small set of pointers to nodes of the graph, one to the "current point of interest" and the rest indicating "yet unsatisfied goals". Each node of the graph indicates one of four things:

1. A terminal symbol is required $\rightarrow * \rightarrow$
2. A non-terminal is required $\rightarrow \langle EC \rangle \rightarrow$
3. A non-terminal has been constructed $\rightarrow (EC) \rightarrow$
4. A production number should be recorded $\rightarrow [3] \rightarrow$

In the graph fragment

$$(EA)^{5} \rightarrow +^{70} \rightarrow [17]^{71} \rightarrow \langle EB\rangle^{72} \rightarrow (EB)^{4}$$

we show that is we have arrived at node 5 (the superscript is used to designate a unique number for each node), we have constructed the non-terminal EA, and that we may build the non-terminal EB on top of the EA by finding a + and an EB in the sentence under analysis. The boxed node 71 indicates the production number 17 should be recorded to indicate how we got from node 5 to node 4.

A graph in this form is derived from the productions of a context free grammar in a mechanical way. We use the BNF form of a context free grammar as described in the ALGOL-60 Report [4], but adding a production number to each production to identify it, and eliminating the use of the vertical bar. Figure 1 gives a grammar for a programming language of medium size in this notation. Figure 2 gives the graph derived from it and economized. The procedure for writing down the graph is the following:

1. Construct one circled node for each non-terminal symbol of the grammar.

2. Construct a node for each terminal symbol which occurs just to the right of the sign ::= . (parentheses denote circled nodes in Figure 2)
3. For each production

   4 $\langle a \rangle$ ::= $bc\ldots d$

   construct a path from the node for b to the node for a, and include along the path new nodes for $c\ldots d$, and one boxed node with the production number in it. (Square brackets denote boxed node in Figure 2)

Thus, the productions

49 <TE> :: = COMMON

50 <TL> :: + <TE>

51 <TL> :: = <TL>, <TE>

produce the graph

$$\begin{tikzcd}
 & & & & \boxed{51} \arrow[d] & \mathrm{TE} \arrow[l] \\
\mathrm{COMMON} \arrow[r] & \boxed{49} \arrow[r] & \textcircled{TE} \arrow[r] & \boxed{50} \arrow[r] & \textcircled{TL} \arrow[ur, bend right, ","']
\end{tikzcd}$$

The graph can be "economized" to save storage and to improve parsing performance (as discussed in Section 3) by combining identical nodes which share the same predecessor or successor. For example the fragment

$$\begin{tikzcd}
\textcircled{EB} \arrow[r] \arrow[dr] & \mathrm{FOR} \arrow[r] & \langle \mathrm{AD} \rangle \arrow[r] & \mathrm{IN} \\
 & \mathrm{FOR} \arrow[r] & \langle \mathrm{AD} \rangle \arrow[r] & \mathrm{TO}
\end{tikzcd}$$

can be collapsed into

$$\begin{tikzcd}
\textcircled{EB} \arrow[r] & \mathrm{FOR} \arrow[r] & \langle \mathrm{AD} \rangle \arrow[r] \arrow[dr, bend right] & \mathrm{IN} \\
 & & & \mathrm{TO}
\end{tikzcd}$$

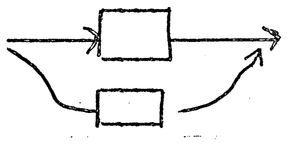
In combining nodes leading to a common successor, a boxed node can be interchanged with its predecessor if it has only one. The graph fragment

$$\begin{tikzcd}
+ \arrow[r] & \langle \mathrm{EB} \rangle \arrow[r] & \boxed{17} \arrow[r] & \textcircled{EB} \\
- \arrow[r] & \langle \mathrm{EB} \rangle \arrow[r] & \boxed{18} \arrow[ur, bend right] &
\end{tikzcd}$$

can therefore be collapsed to

$$\begin{tikzcd}
+ \arrow[r] & \boxed{17} \arrow[r] & \langle EB \rangle \arrow[r] & \textcircled{EB} \\
- \arrow[r] & \boxed{18} \arrow[ur, bend right] & &
\end{tikzcd}$$

Incidentally, if this process leads to a configuration



we have identified an ambiguity in the language described by the grammar! The graph of Figure 2 has been economized in this way, although this was done by hand, and a few possibilities were missed! See if you can find them.

3. MINIMUM PROCESS TREES

As discussed in "Multiple-Track Programming" [5], the parsing process can be characterized by process trees where each node of the tree indicates a state of the parsing process and where local ambiguity is reflected in having several nodes at the same distance from the root. A minimum process tree is one in which no nodes at a fixed distance from the root can be eliminated without knowledge of what happens further from the root. The discussion in [5] indicates that for languages suitable for human use, psychological considerations lead us to expect only a few nodes at a given distance from the process tree root, because the presence of more than one node indicates genuine ambiguity in the vicinity of the node.

We shall use the "condensed bracketing" notation introduced in [5] to describe parses. This notation is derived from the conventional parse bracketing notation by merging brackets whose left ends occur at the same place. The conventional and condensed bracketing of a small sentence according to the graph of Figure 2 is shown in Figure 3. In Figure 3 we use production numbers to denote the origin of the bracket. In the condensed bracketing, the number for the inner most original bracket is written first, next inner most next, etc. so that we may derive the conventional diagram from the condensed diagram.

In the condensed bracket notation, each bracket corresponds to a path through the graph from the entry point for the symbol at its left end to the circled node for its goal. The production numbers indicate the square boxes parsed along the way, and identify the path uniquely. Another way of identifying this path is to write down the node number of each node which causes us to match a terminal symbol or set a new goal under the symbol we matched or used to begin the new goal. (Actually ambiguities of a trivial kind such as those introduced by duplicated productions will not allow this second notation method to distinguish the different paths, but this is a minor restriction). The bracketing using this convention is shown in Figure 4. Node numbers have been written above the bracket lines to avoid confusing them with production numbers.

Using the notation of Figure 4, we can describe each node of the process tree for a parse by writing down in order, the node numbers under a symbol (taking the next one to the left where one is missing). Thus the (minimum) process tree for the sentence of Figures 3 and 4 is shown in Figure 5. This process tree reflects our inability to determine which use of the < sign is intended without looking at the symbol which follows it.

We now describe the construction of a *bottom-up* parsing program which will always use a minimum process tree. We shall use as our departure point the algorithm from [5], which leads eventually to the fast parse algorithm given in Section 5.

The algorithm of [5] must be modified to avoid wandering off into areas of the graph which do not lead to the current goal. This mistaken meandering could occur at node 4 of Figure 2, for example with the goal EL. The algorithm of [5] will go off to (among other places) nodes 127, 7, 139, 9, and 140 to match a comma as well as going to node 152 to match the comma. In fact an expression

X + Y, P*Q, M[N]

is both a legitimate EL and FL and the algorithm of [5] will construct parses for both, even though an FL can never turn into an EL which is the current goal. The remedy for this problem is to construct a connectivity matrix to show whether or not a path exists between two nodes of the graph.

The algorithm of Section 5 uses such a matrix, and is thereby forced to stick to the minimum process tree. Each node of the process tree will contain a set of numbers which identify the current stack of goals (in terms of the syntax graph nodes where the goals were set) and a pointer to the entry point of the graph for the current symbol if a new goal has been set. As each new symbol is read, it either matches a required terminal, or leads to a required non-terminal at the current point in each parse, or the parse is not continued. As goals are reached every step taken in the syntax graph is checked against the connectivity matrix to insure that the successor node is still on a path to the goal it must reach. Clearly there will be one such successor node or the node before it could not have been on a path to the goal. So each of the syntax graph nodes listed in a process tree node leads to the goal requested by node listed below it, and the bottom one leads to the final goal. The process for terminating brackets is constructed so that each of these pending nodes has a possibility of working its way to its goal by finding more symbols in the sentence. We can therefore not eliminate any nodes of the process tree without reading more symbols. Since in a context free grammar, the validity of drawing a parse bracket is determined entirely by what the bracket embraces, we are assured that nothing will allow us to reject a node of the process tree except symbols further on in the sentence. So the process tree will be minimal.

A *top-down* parser cannot use the syntax graph directly, but must have it broken into a set of graphs, one for each non-terminal, telling how to construct the non-terminal. This is accomplished by starting at the circled node for a non-terminal and working backwards. When another circled node is reached, it is changed into a non-circled node, and the backward moving process is terminated. All circled nodes reached this way are collected together as successors of an entry node for the non-terminal. Loops in the graph are left intact, however, to avoid the *left-recursive-production* problem for *top-down* parsing. The resulting *top-down* subgraphs for *PG, ED, EC, EB, EA, AD,* and *NL* are shown in Figure 6. In this graph we have kept the same node numbers as in Figure 7, except where circled nodes have been turned into nodes requesting non-terminals. These nodes have a new number appended to the old one to make the numbers unique.

Figure 7 shows the parse and Figure 8 the process tree for the parse of the same sentence analyzed with the *bottom-up* parse program in Figure 4. Brackets have not been condensed here as the *top-down* algorithm does not produce them in condensed form except when going around a loop, and no loops are traversed in this example. The fact that the *top-down* parser does not produce the condensed bracketing leads to more branching in the process tree, but as we shall discuss in Section 4, less work is required to derive the average node, so we expect the total amount of work required for the top-down and bottom-up programs to be roughly the same in the absence of further optimization. For the bracketing given, however, the process tree will be a minimum one. The argument is essentially the same as for the bottom-up case. A connectivity matrix is not required for the top-down parser, since all nodes in the tree for a given non-terminal lead to its circled node; we have constructed the tree this way.

Parsing programs for bounded-context grammars all have a process tree with no branching, since the point of using such a grammar for these programs is precisely to eliminate the branching and hence the requirement for backtracking or multiple-tracking.

4. NODE DERIVATION TIME

We have established that for reasonable languages, the top-down and bottom-up algorithms we are discussing will not have a process tree substantially bigger than that for bounded-context parsers. However, the node derivation time for both of these algorithms is substantially longer than that for at least the fastest of the bounded-context systems. We shall now show a simple modification for the bottom-up algorithm which makes its node derivation time compare well with bounded-context parsers. We will compare it, in particular, with the operator precedence parser of Floyd [7] since that is one of the fastest of these programs. We have not discovered a similar optimization for the top-down method.

The unoptimized version of the BU algorithm must visit a large number of nodes of the syntax graph when deriving some new nodes of the process tree. For example, in deriving the second two nodes of Figure 5 (those for the < sign) we visit all nodes which can be reached from node 15 without passing through a "terminal required" or "non-terminal required" node. The nodes visited are 155, 13, 68, 65, 156, 106, 103, 110, 6, 102, 5, 94, 4, 93, 3, 21, 2, 159, 163, 32, 26, 70, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 22, and 46, a grand total of 34 of the 160 odd nodes in the graph. (The top-down program will visit pretty much the same list of nodes eventually). At each of these nodes (except the boxed ones where we just record the number in the box) we ask if we have arrived at a goal (if the node is circled) or whether the symbol at hand matches one required or will begin a required non-terminal. The answer to these questions will be yes only at a very few nodes. In the case discussed above only nodes 81 and 22 yield a "yes" answer.

The modification of the algorithm is essentially to change the search through the nodes of the syntax graph to a table lookup. The table we need is one which answers the question "if at node i, and having the symbol j, what nodes should next be considered?" A table to answer this question in complete generality would be large, since one dimension is the number of nodes (about 160 here) and the other the number of possible terminal symbols (about 35 here).

If we restrict the class of nodes at which we are to take this "giant step" to circled nodes, however, we achieve almost all the saving. The only non-circled nodes where branching occurs in Figure 2 are 130, 32, and 27, and the degree of branching is small at these nodes. Even so, we could include them in the table lookup process by introducing our artificial circled node at these places.

The table we need to store now can have an entry for each symbol telling where that symbol occurs after any circled node, and a bit pattern telling which circled nodes the symbol can follow. Since some symbols can follow circled nodes in several places (a comma, for example, occurs after circled nodes at 156, 140, 148, and 152) we will have to link together entries for such multiple occurances. It happens in Figure 2 that there are no "non-terminal required" nodes following circled nodes. In general there could be such occurances, of course, but such nodes are then listed for each terminal which can begin the requested non-terminal. (Whether a terminal begins a non-terminal can be determined from the connectivity matrix). The details of these tables are discussed in Section 5 in conjunction with the detailed description of the optimized BU algorithm.

The other type of question asked at circled nodes is "has the current goal been reached?" This question can be answered by a connectivity matrix too, namely, one which tells which circled nodes are connected to which others by paths going only through circled (and boxed) nodes.

With these tables in use, only one node is examined in moving from one node of the process tree to the next, except when goals are satisfied. In this case one more node for each satisfied goal need be examined. This processing, however, it almost identical in its requirement to that required to move from one symbol to the next in Floyd's algorithm.

The optimized BU algorithm will therefore be slower than bounded-context parsers only by the branching factor of the process tree. This means that it will equal their performance on grammars that can be used with those methods, and in any event will not be significantly worse on other grammars, which are designed (as they should be to make the language easy for people to use) to allow only a modest amount of local ambiguity.

The optimized BU algorithm will not produce output identical to that for the unoptimized version. In particular, because boxed nodes are skipped over when the tables are employed, brackets which cover only one other bracket or symbol will not be put onto the parse. If it is desired to have these brackets in the output, it will be required to store the list of boxed nodes passed over in moving from a circled node to the nodes picked out of the tables. It is easy enough to include such a provision, but it is the author's experience that most of these brackets are ignored in post-parse processing anyway, and it is a small inconvenience to do without them altogether.

It is unclear whether similar optimization could be applied to the top-down algorithm. Could this be done, however, it is unlikely that the result would be any better than with the BU algorithm, and so we recommend the TD approach be rejected altogether in favor of the optimized BU program.

5. THE FAST PARSE ALGORITHM

The fast parse program is described in a global flow diagram in Figure 9, and in detail in Figure 10. Figure 11 describes in detail the section for processing a circled node in the unoptimized BU program. The fast parse program has been made from this BU program by replacing only this section. The BU program is the bottom-up multiple-tracking program of [5], but tailored, as suggested there, to the multiple-tracking framework, and augmented to use the connectivity matrix. The arrays used by the programs are described in Figure 12. The syntax graph is contained in arrays POINT, LINK, TYPE, and the connectivity matrix in BITS. The words describing one node are linked together in LINK. The first of these for every node is contained in the beginning of the arrays, and the POINT value is the index of the characters for the terminal or non-terminal in DI (or a production number for boxed nodes). The rest of the point values for a node occur in the last part of the arrays and contain pointers to successors nodes. It has been arranged that the non-terminals have directory entries which are the same as the node number for the circled node for the non-terminal. Thus, all non-terminals are identified by a small number. This number is used as a shift count for bits in the connectivity matrix BITS. If node I reaches circled node J then the Jth bit of BITS [I] (from the right) will be 1, or BITS [I] $\wedge$ (1 LS J) will be non-zero. The TYPE array indicates the type of a node as follows

TYPE [I] = 0 means the node requests a non-terminal

TYPE [I] = 1 means the node requests a terminal symbol

TYPE [I] = 2 a circled (non-terminal reached) node

TYPE [I] = 3 a boxed (output) node.

DI contains the characters for the words recognized by the lexical analyzer. The initial DI is DIN words long and contains at its beginning the names of the non-terminals, then the terminal symbols. As the parsing proceeds, names occurring in the program being parsed are added to the end of DI. These are all taken by the parser to be the terminal symbol NAME, but then indices in DI are placed in the output stream for use after the parsing process. ENTRY gives the graph entry point for every terminal symbol which has one.

NODE, NLINK, and NBITS store the information required by the circled node section of the fast parser. The beginning of these arrays is parallel with the DI and ENTRY arrays so there will be first entries for the non-terminals and then entries for the terminals.

The entries for the non-terminals store the "circled node connectivity matrix". If NBITS [I] $\wedge$ (1 LS J) is non-zero, then circled node J can be reached from circled node I by passing only through circled or boxed nodes. Thus, it answers the question "if at node I, have I actually made the non-terminal for node J?"

The entries for the terminal symbols give the graph locations where the terminals are called for (either directly or because they begin a non-terminal which is called for) after circled nodes. The NODE entry tells the node at which the terminal is requested. The NBITS entry tells what circled nodes preceed the node NODE. Thus for the terminal symbol I, and a circled node J, if NBITS [I] $\wedge$ (1 LS J) is non-zero then the terminal symbol I is requested at node NODE [I] and this node can be reached from J by passing through only circled or boxed nodes. Since a terminal symbol can occur at several places in the graph, several entries for each terminal symbol may be required.

If more than one entry is necessary, the extra ones are put on the end of the arrays and linked with NLINK. In the syntax graph of Figure 2, the symbol < has the nodes 22 and 81 listed in node. NBITS will be 100000001100000 for node 81 (indicating that node 81 comes after nodes 5,6, and 14) and 100000001000000 for node 22 (indicating it comes after nodes 6 and 14).

The contents of these arrays for the graph of Figure 2 is given in Appendix 1. The node numbers given as superscripts in Figure 2 correspond to the node numbers in POINT, etc.

The parses are kept in array P. P has been set to hold 11 parses of length 30 in the program of Figure 10. Each parse has

P[I]	=	a pointer to the current node of the graph
P[I+1]	=	a pointer to the last item of output
P[I+2]	=	the word of this parse containing the last goal
P[I+3]		pairs of words containing tentative
P[I+4]		output (first word) and goals (second word)

Thus DI [POINT [P[I]]] is the name stored at the node now being processed by parse I. DI [POINT[P[I+P[I+2]]]] is the name of the current goal.

CUR holds the CN indices in P of all the current parses, and NEW holds the NN indices in P of parses which have been extended, and hence will be used for the next symbol.

OUT and OUTL hold the output for the current parse. Because we have moved some boxed nodes back in the graph, the production number in a boxed node cannot be put directly in the output, or the final result will be in the wrong order. For example, the production number 9 at node 47 must not be put into the output until the outputs from the EC called for at node 25 have been put there. Therefore when a boxed node is parsed, its production number is put into the goal stack at (P[I+3], P[I+5]...) until the next circled node is reached at the same goal level. Then the production number is put into OUT and OUTL is used to link to the last item of output put into OUT for this parse. The links are necessary since all of the output from all parses is kept in OUT until only one parse is found valid. At that time the output is removed from OUT and either stored compactly in a "final output" array or passed on to generating programs immediately.

```
 1 <PG>::=<ED>
56 <ED>::=<ED> ; <EC>
 2        <EC>
 3 <EC>::=<AD> '<' - <EC>
 4        <EB> '=' '>' <EC>
 5        <EB> '=' '>' <EB> ELSE <EC>
 6        <EB> FOR <AD> IN <EB>, <EB>,<EB>
 7        <EB> FOR <AD> TO <EB>
 8        <EB> FOR <AD> FROM <EB>
 9        <NAME> ':' <EC>
10        GO TO <NAME>
11        SUBR <NAME> ( <NL> ) IS <EC>
12        PRINT <FL>
13        READ <FL>
14        <NL> ARE <TL>
15        <NL> IS <TL>
16        <EB>
17 <EB>::=<EA> + <EB>
18        <EA> - <EB>
19        <EA> * <EB>
20        <EA> / <EB>
21        <EA> AND <EB>
22        <EA> '<' <EB>
23        <EA> '>' <EB>
24        <EA> '=' <EB>
25        <EA> OR <EB>
26        <EA> LS <EB>
27        <EA> RS <EB>
28        <EA>
29 <EA>::=NOT <EA>
30        ( <ED> )
31        <AD>
32        <NAME> . <NAME>
33        <NAME> ( <EL> )
34 <AD>::=<NAME>
35        <NAME> [ <EC> ]
36 <FE>::=OCT <EB>
37        IGR <EB>
38        STG <EB>
39        FILE <FN>
40        /
41        <EB>
42 <FN>::=<NAME>
43        <NAME>.<NAME>
44        <NAME>.<NAME>[<NAME>,<NAME>]
45        <NAME>[<NAME>.<NAME>]
46 <FL>::=<FE>
47        <FL>,<FE>
48 <TE>::=<NAME> LONG
49        COMMON
50 <TL>::=<TE>
51        <TL>,<TE>
52 <EL>::=<EB>
53        <EL>,<EB>
54 <NL>::=<NAME>
55        <NL>,<NAME>
```

FIGURE 1: THE GRAMMAR FOR THE SYNTAX GRAPH IN FIGURE 2.

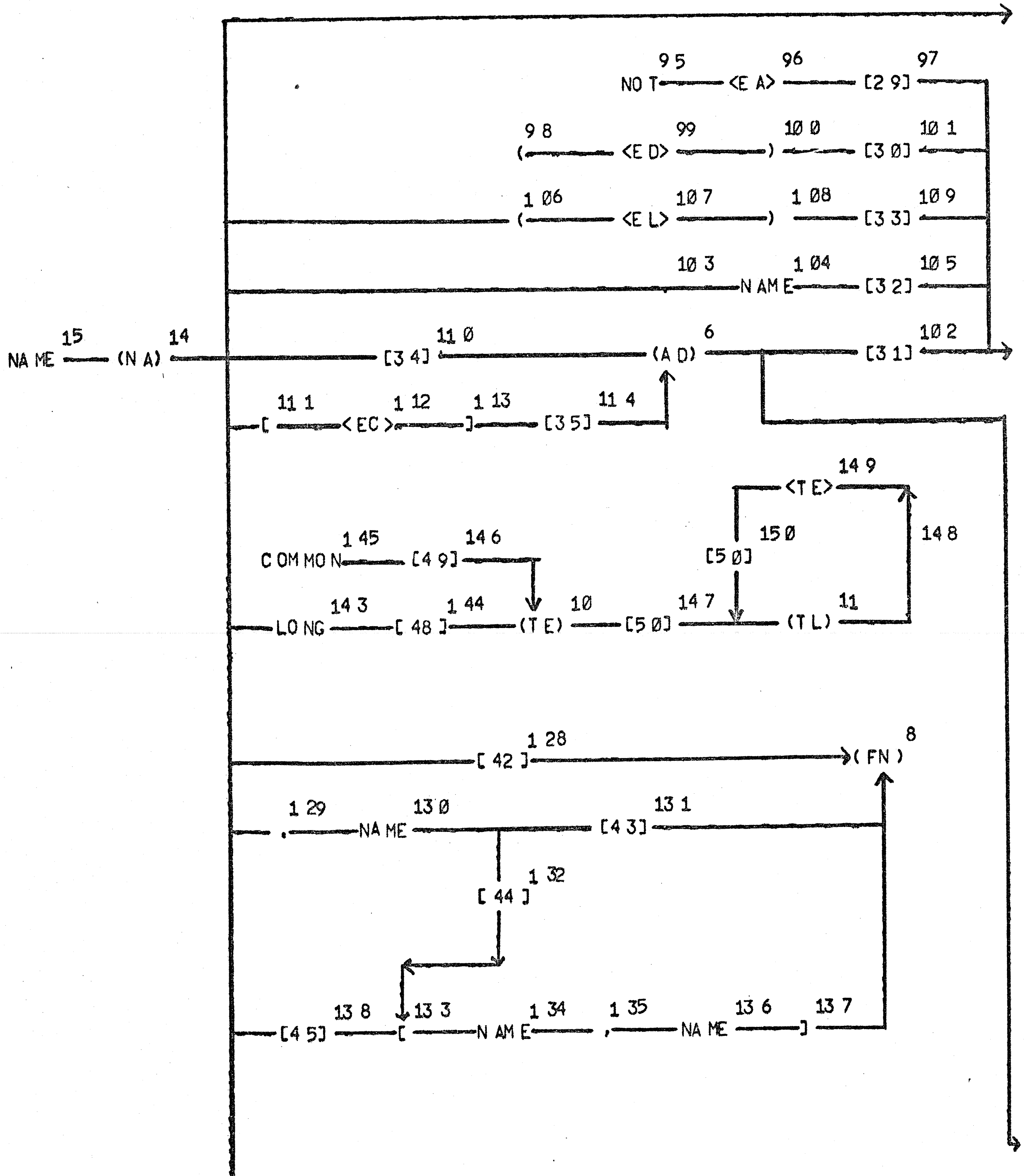


FIGURE 2(1): THE SYNTAX GRAPH FOR PRODUCTIONS OF FIGURE 1

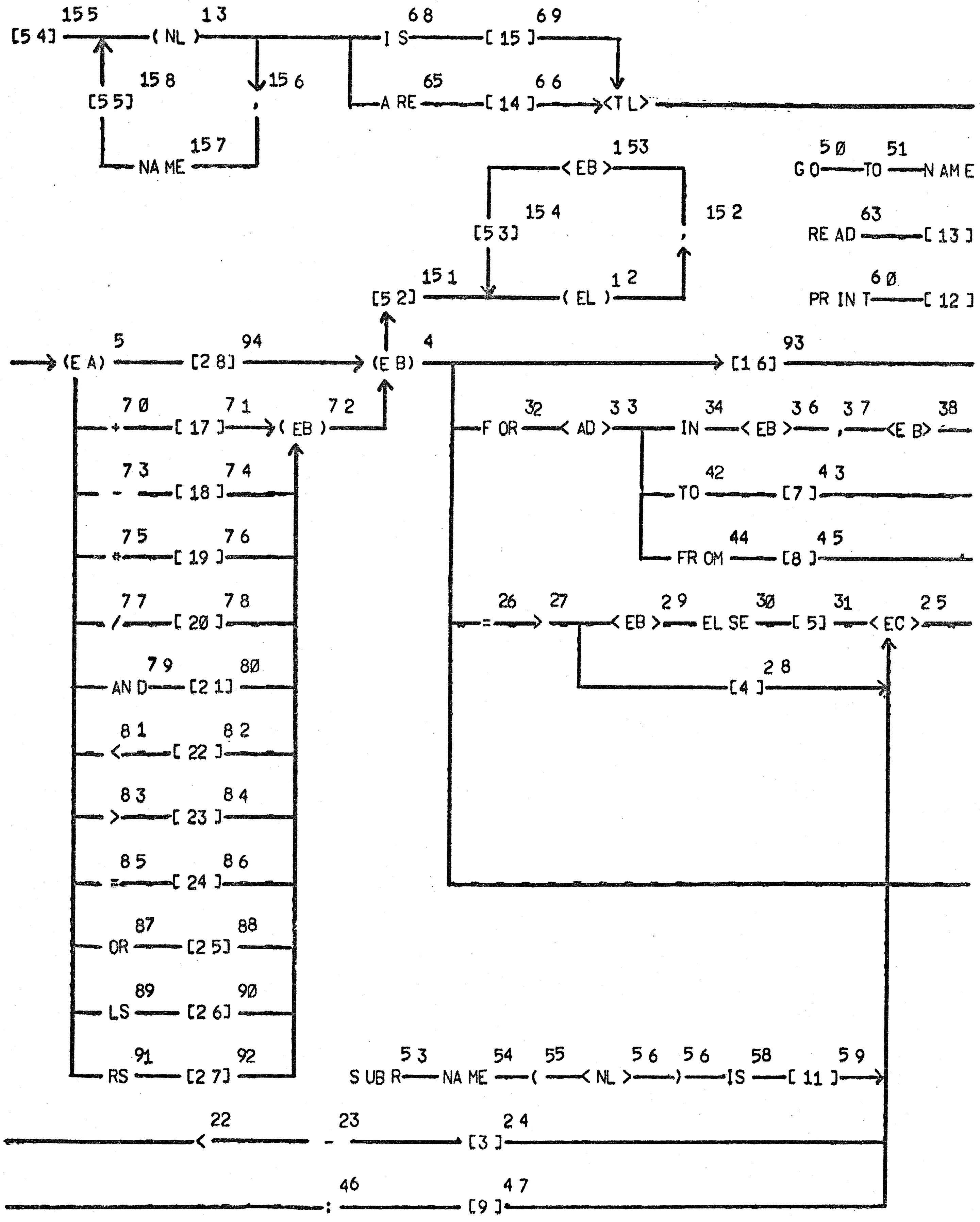
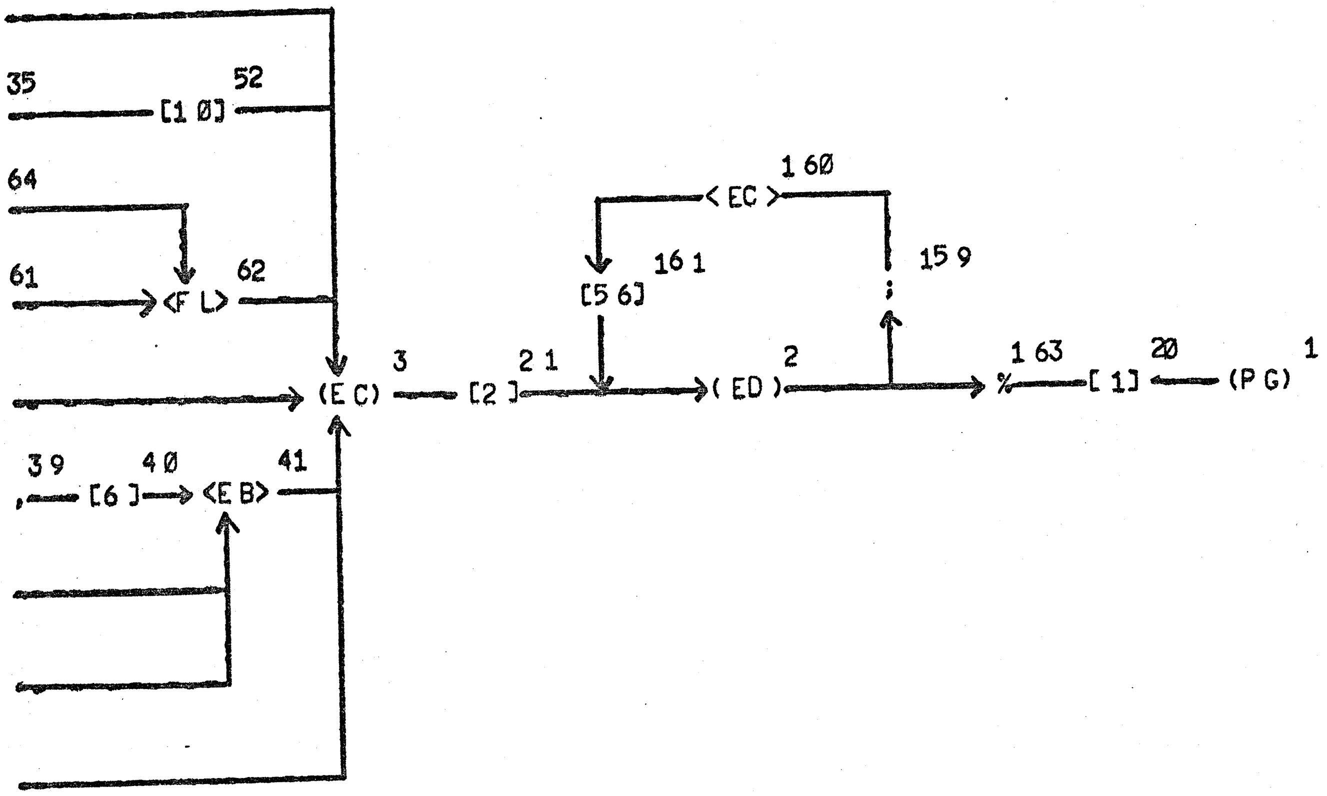


FIGURE 2(2): (HOOK THIS ON TO THE RIGHT OF THE LAST PAGE)



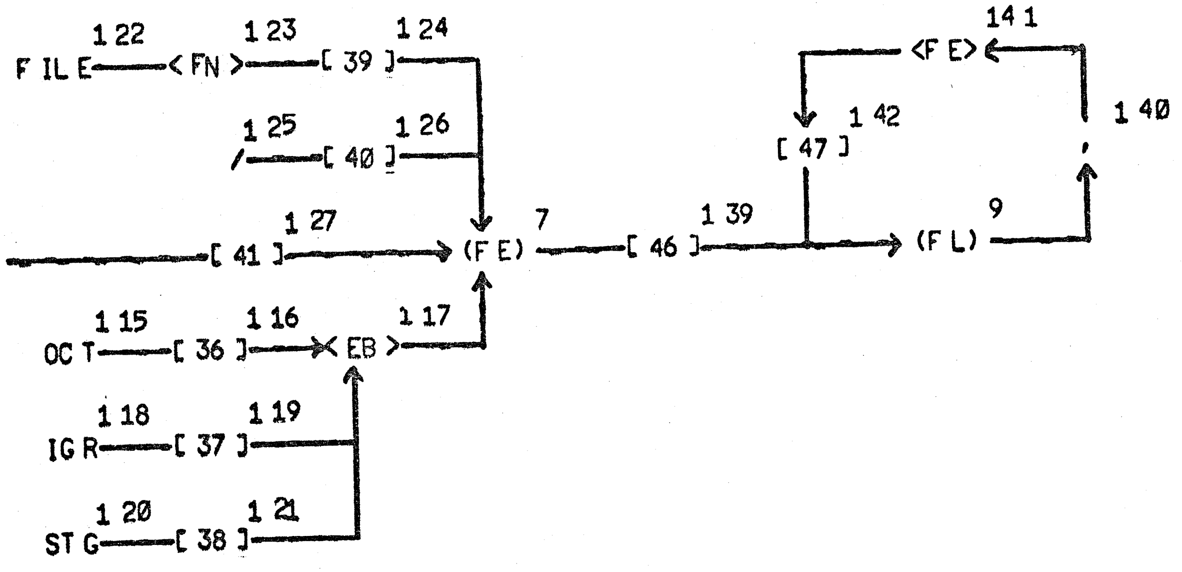
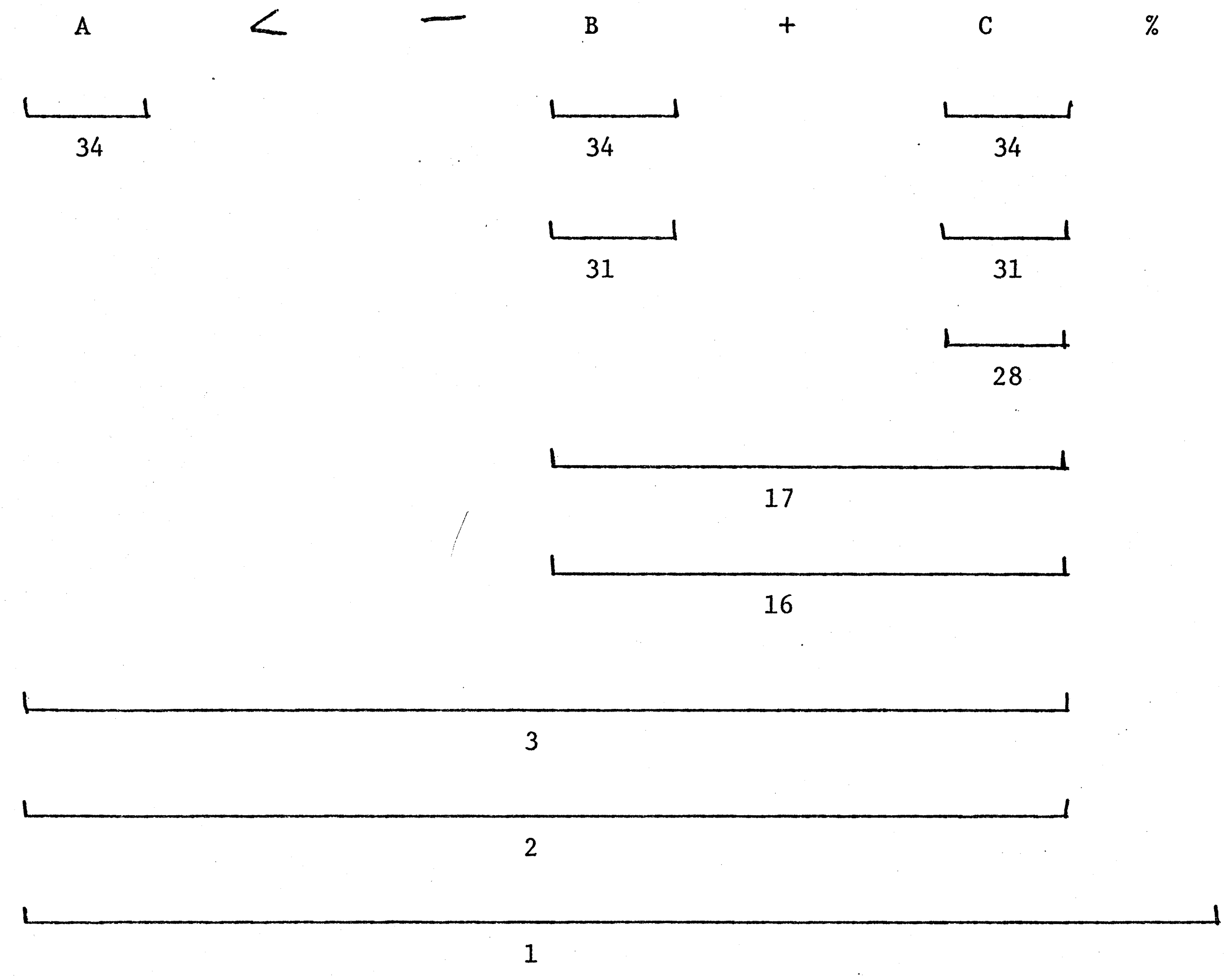
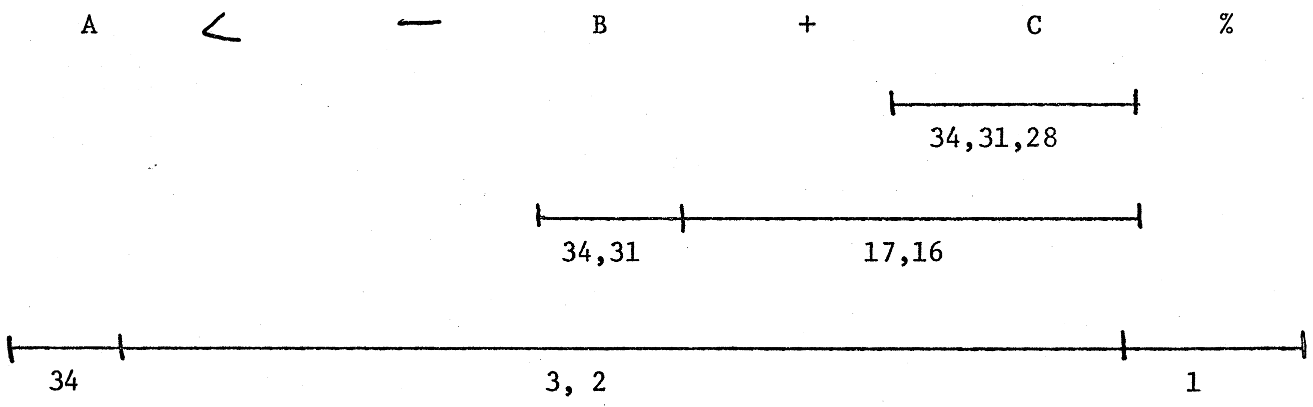


FIGURE 2(3): (HOOK THIS ONTO THE RIGHT OF THE LAST PAGE)



CONVENTIONAL BRACKETING



CONDENSED BRACKETING

FIGURE 3. CONDENSED PARSE NOTATION

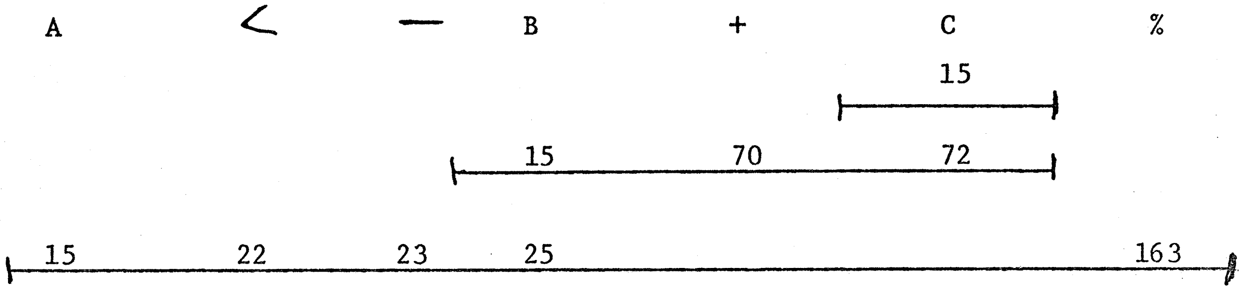


FIGURE 4. CONDENSED PARSE NOTATION WITH GRAPH PATHS INDENTIFIED FROM GOAL SEEKING NODES.

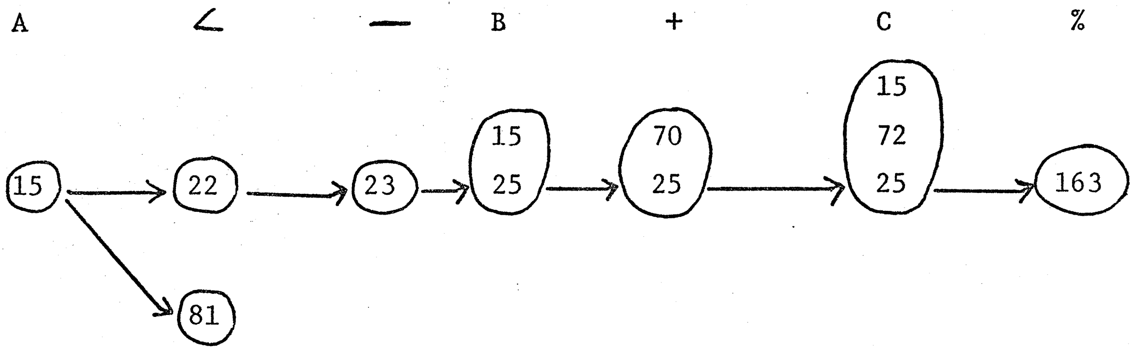


FIGURE 5. THE MINIMUM PROCESS TREE FOR THE PARSE OF FIGURES 3 and 4.

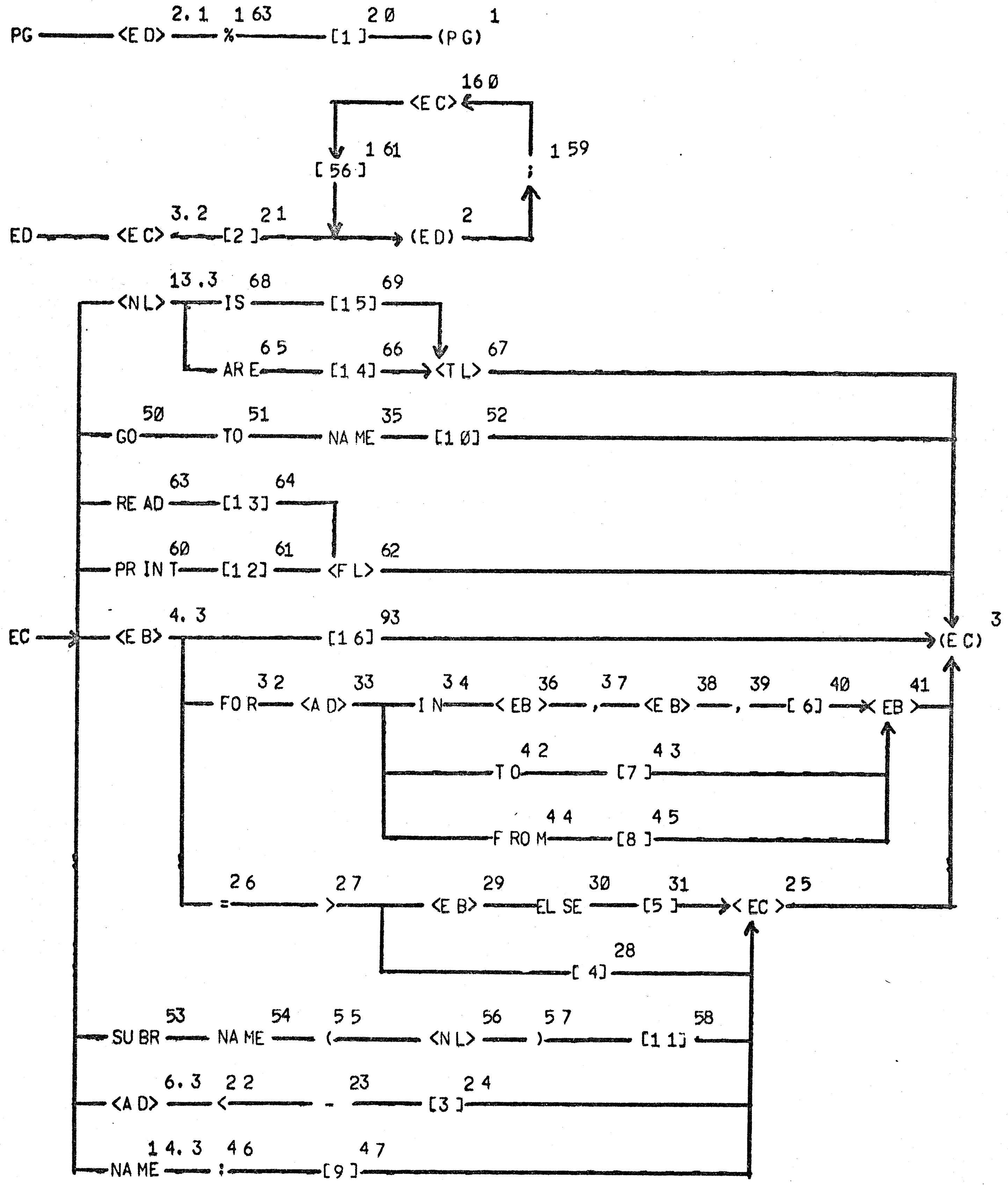


FIGURE 6(1). PARTS OF THE TOP DOWN GRAPH FOR FIGURE 2.

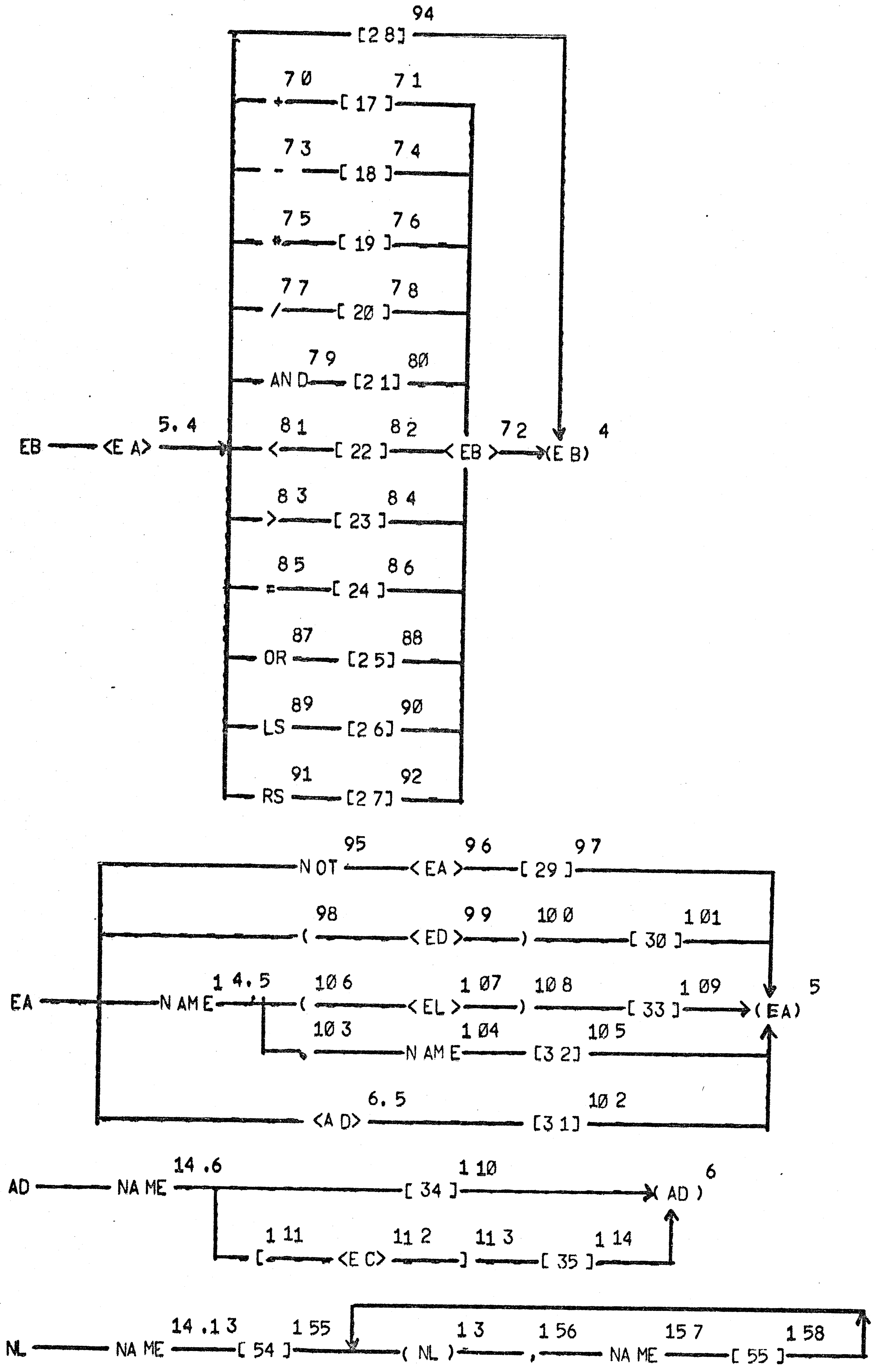

94
[28]
70 71
+ [17]
73 74
- [18]
75 76
* [19]
77 78
/ [20]
79 80
AND [21]
EB <EA> 5.4
81 82 72
< [22] <EB> (EB) 4
83 84
> [23]
85 86
= [24]
87 88
OR [25]
89 90
LS [26]
91 92
RS [27]
95 96 97
NOT <EA> [29]
98 99 100 101
(<ED>) [30]
EA NAME 14.5
106 107 108 109
(<EL>) [33] (EA) 5
103 104 105
NAME [32]
6.5 102
<AD> [31]
AD NAME 14.6
110
[34] (AD) 6
111 112 113 114
[<EC>] [35]
NL NAME 14.13 155
[54] (NL)
13 156 157 158
, NAME [55]

FIGURE 6 (2)

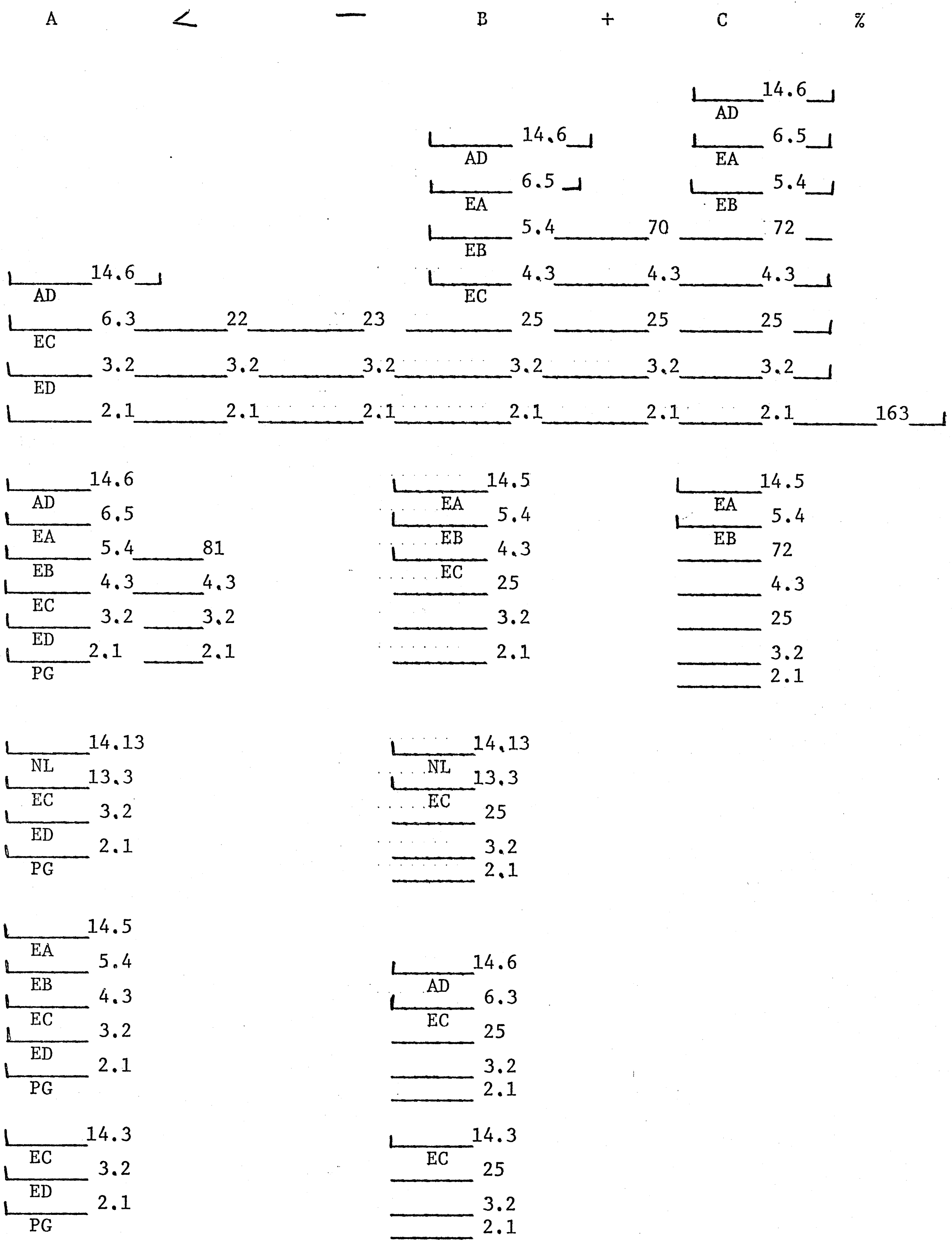


FIGURE 7. TOP-DOWN PARSE USING THE GRAPH OF FIGURE 6.

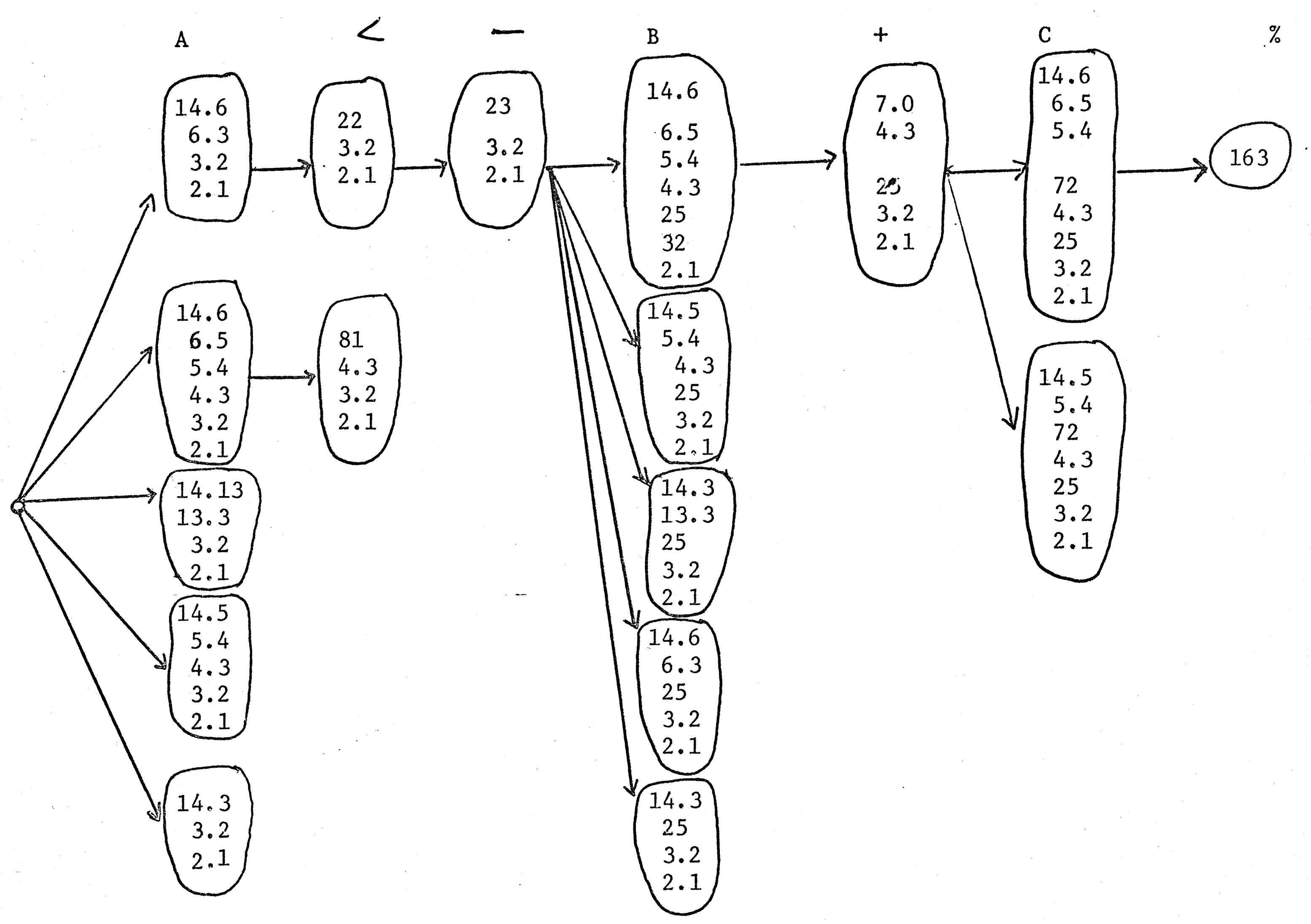


FIGURE 8. THE PROCESS TREE FOR THE PARSE OF FIGURE 7.

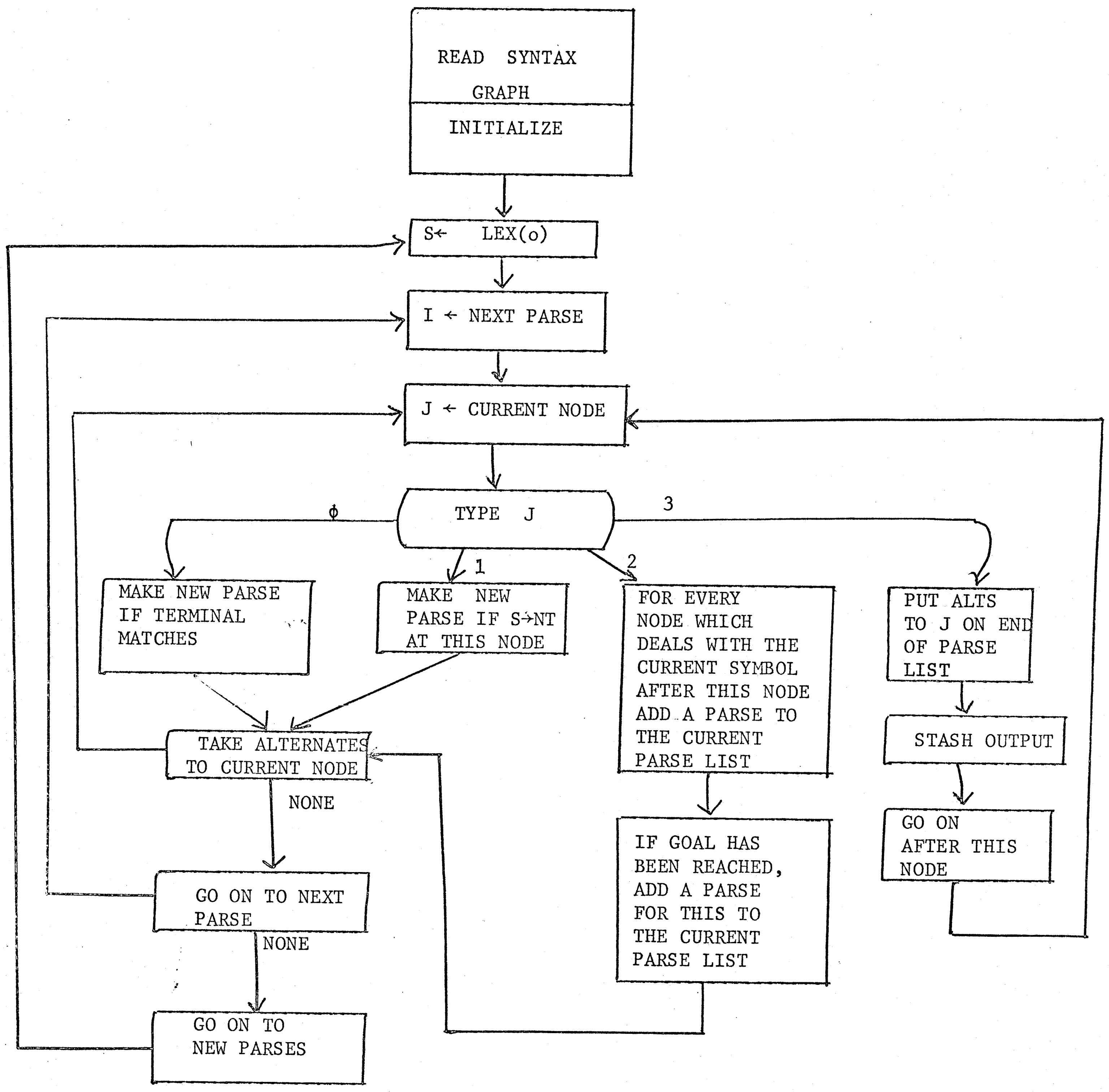
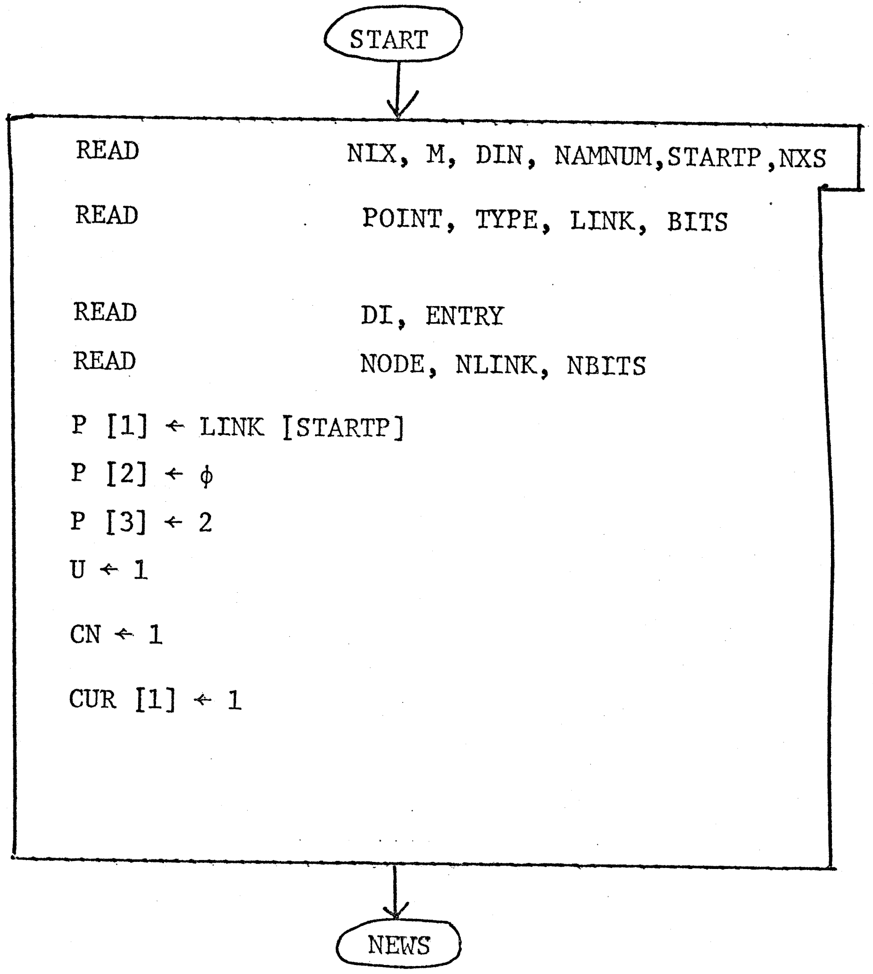

READ SYNTAX GRAPH
INITIALIZE
S← LEX(o)
I ← NEXT PARSE
J ← CURRENT NODE
TYPE J
ϕ
1
2
3
MAKE NEW PARSE IF TERMINAL MATCHES
MAKE NEW PARSE IF S→NT AT THIS NODE
FOR EVERY NODE WHICH DEALS WITH THE CURRENT SYMBOL AFTER THIS NODE ADD A PARSE TO THE CURRENT PARSE LIST
PUT ALTS TO J ON END OF PARSE LIST
TAKE ALTERNATES TO CURRENT NODE
STASH OUTPUT
NONE
IF GOAL HAS BEEN REACHED, ADD A PARSE FOR THIS TO THE CURRENT PARSE LIST
GO ON AFTER THIS NODE
GO ON TO NEXT PARSE
NONE
GO ON TO NEW PARSES

FIGURE 9. GLOBAL DIAGRAM OF THE FAST PARSE PROGRAM.



```
READ            NIX, M, DIN, NAMNUM,STARTP,NXS
READ            POINT, TYPE, LINK, BITS

READ            DI, ENTRY
READ            NODE, NLINK, NBITS
P [1] ← LINK [STARTP]
P [2] ← φ
P [3] ← 2
U ← 1
CN ← 1
CUR [1] ← 1
```

FIGURE 10 (1) THE FAST PARSE PROGRAM: INITIALIZATION

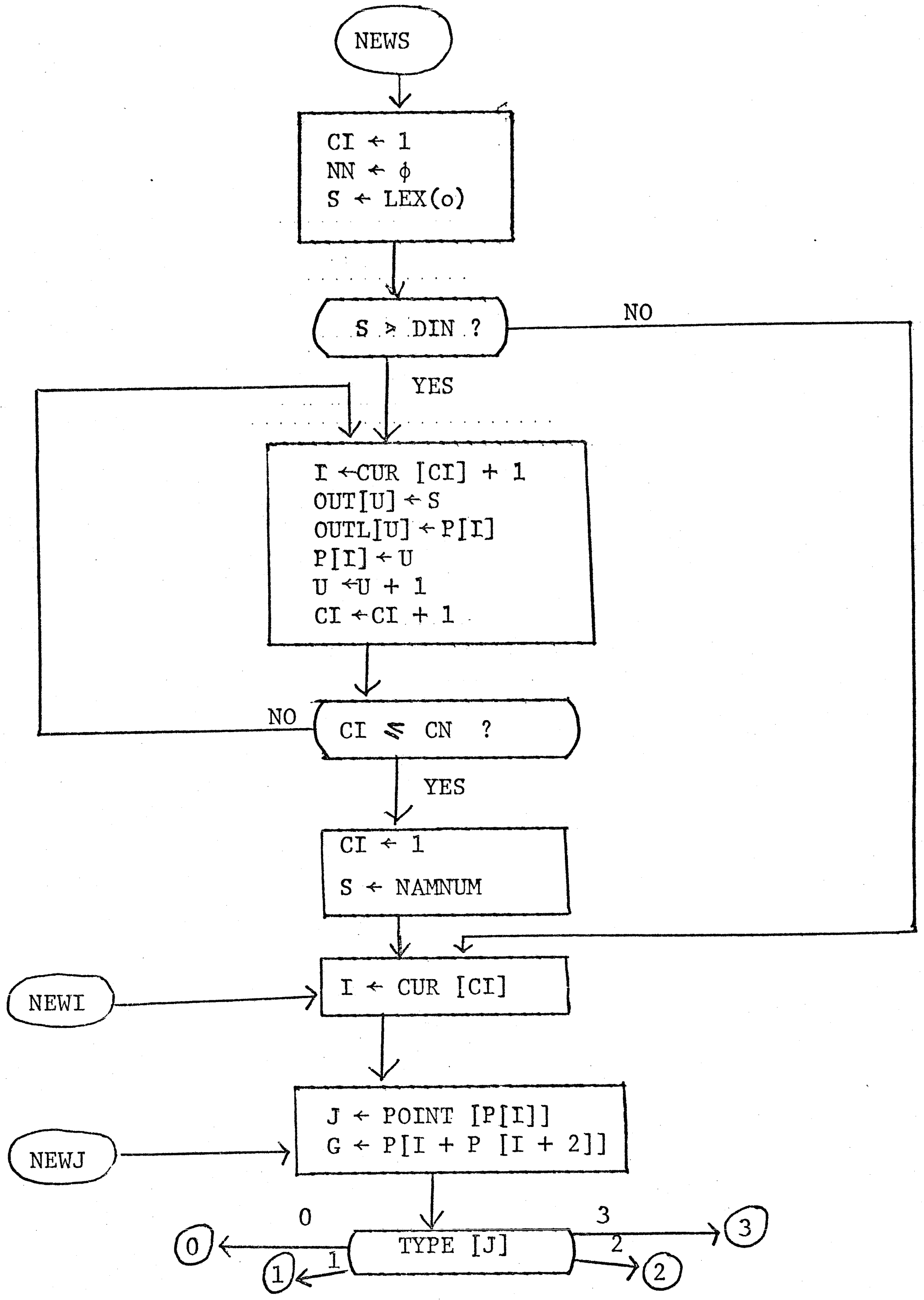


FIGURE 10 (2) THE FAST PARSE PROGRAM: PROCESSING A NEW SYMBOL, NEW PARSE, NEW NODE

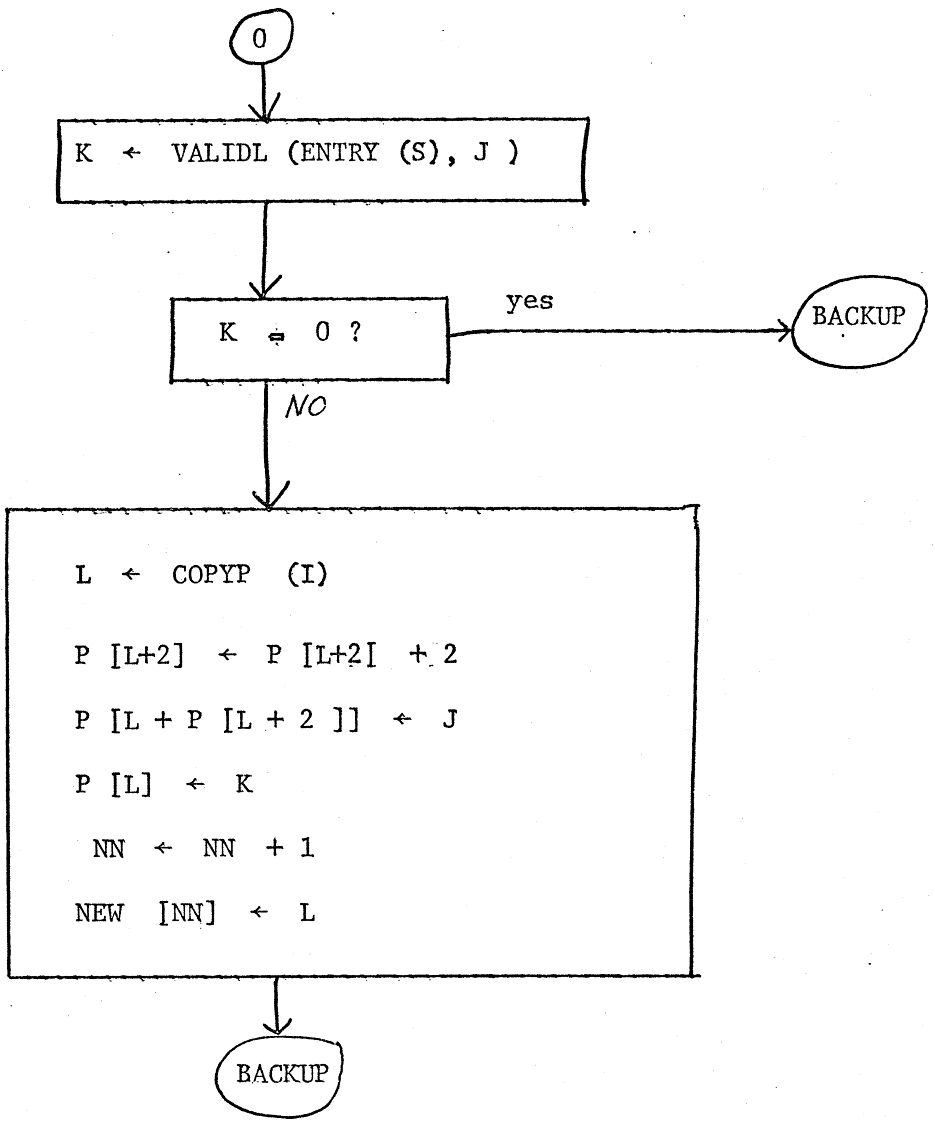


FIGURE 10 (3) THE FAST PARSE PROGRAM: PROCESSING A NODE WHERE A NON-TERMINAL IS REQUIRED.

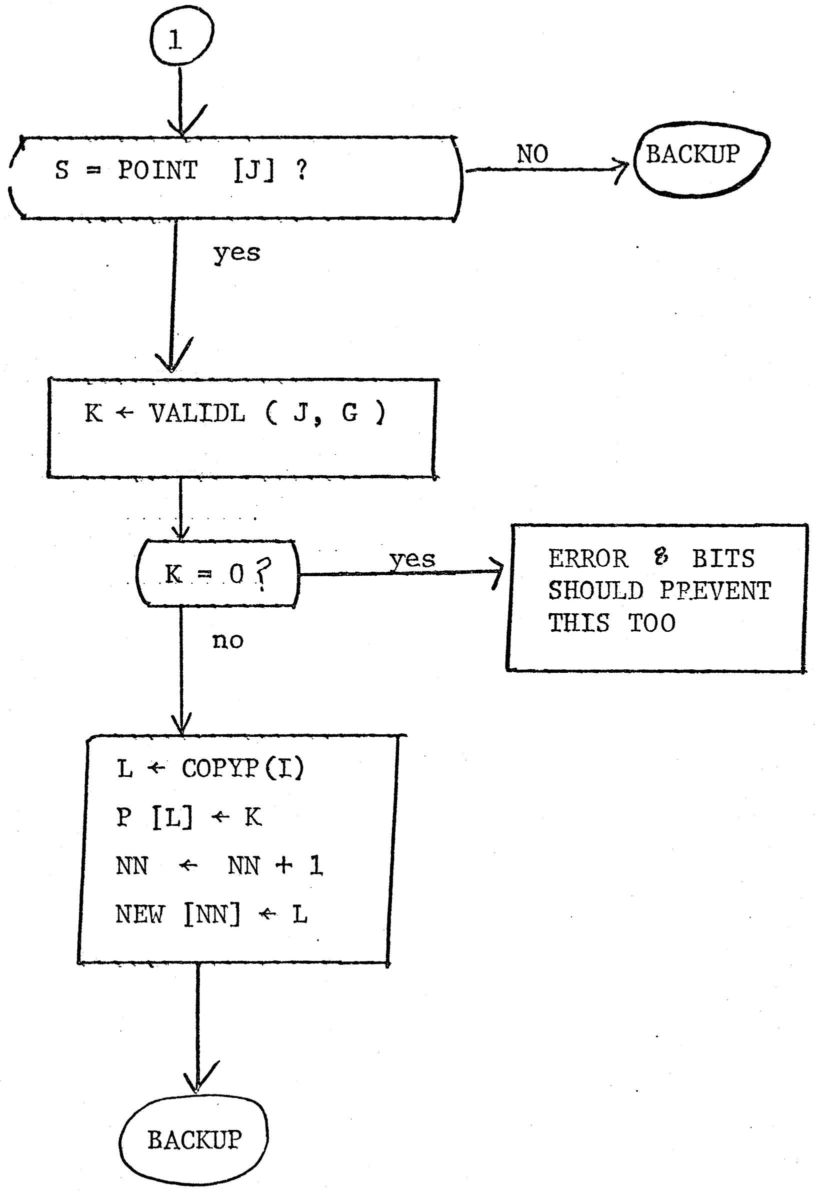


FIGURE 10 (4) THE FAST PARSE PROGRAM: PROCESSING A NODE WHERE A TERMINAL SYMBOL IS REQUIRED.

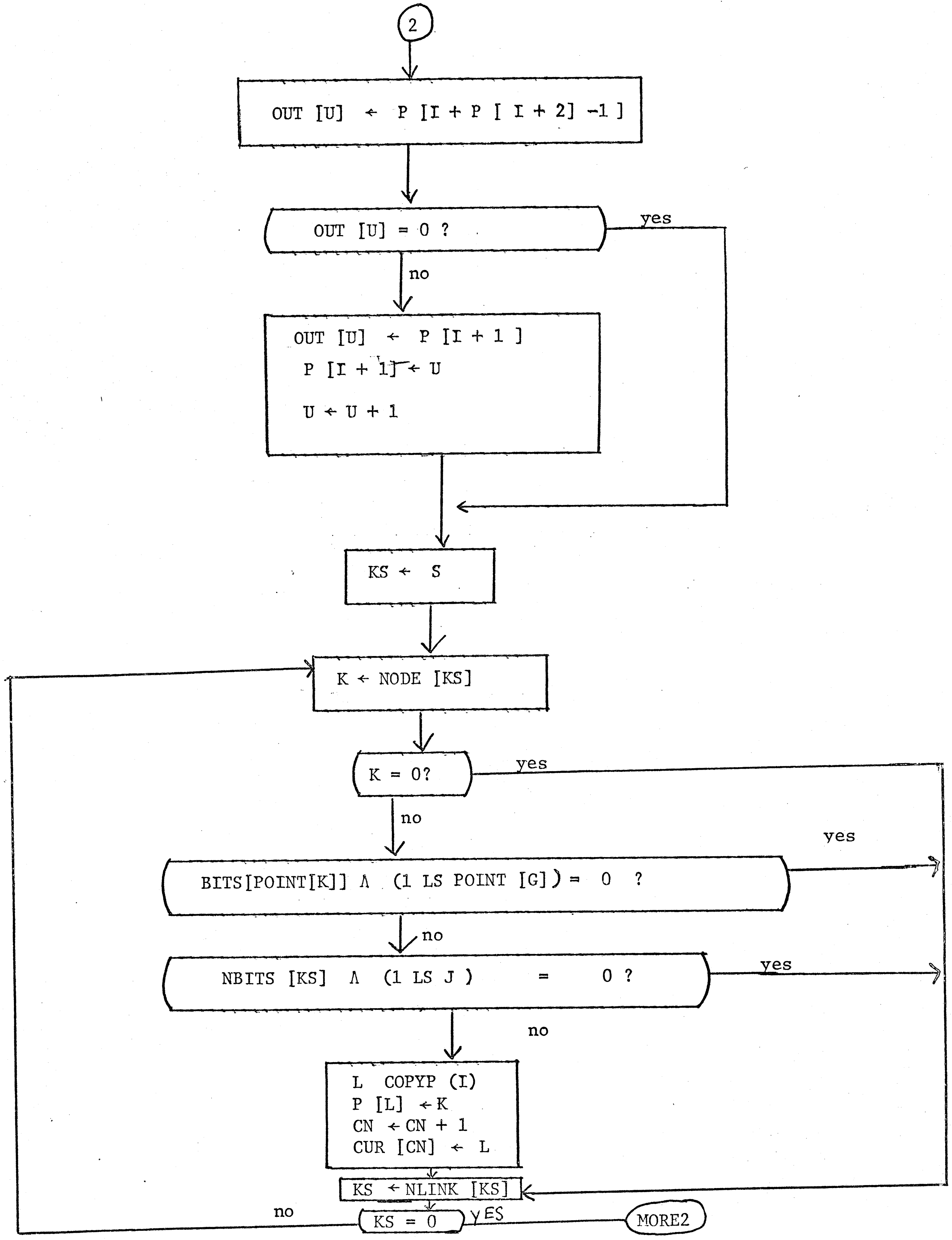


FIGURE 10 (5) THE FAST PARSE PROGRAM: PROCESSING A NODE WHERE A NON-TERMINAL IS CONSTRUCTED.

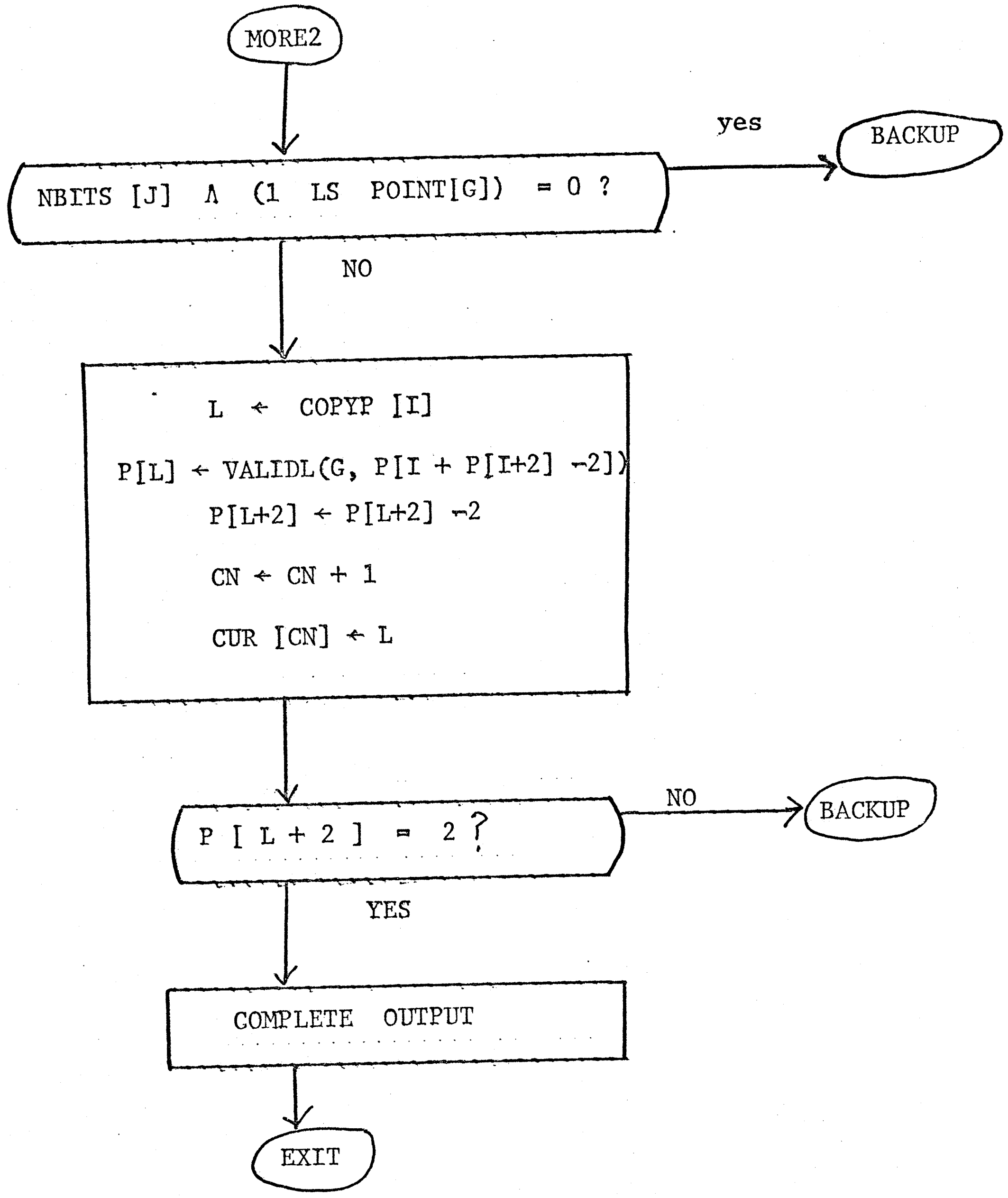


FIGURE 10 (6) THE FAST PARSE PROGRAM: PROCESSING A CIRCLED NODE (CONTINUED)

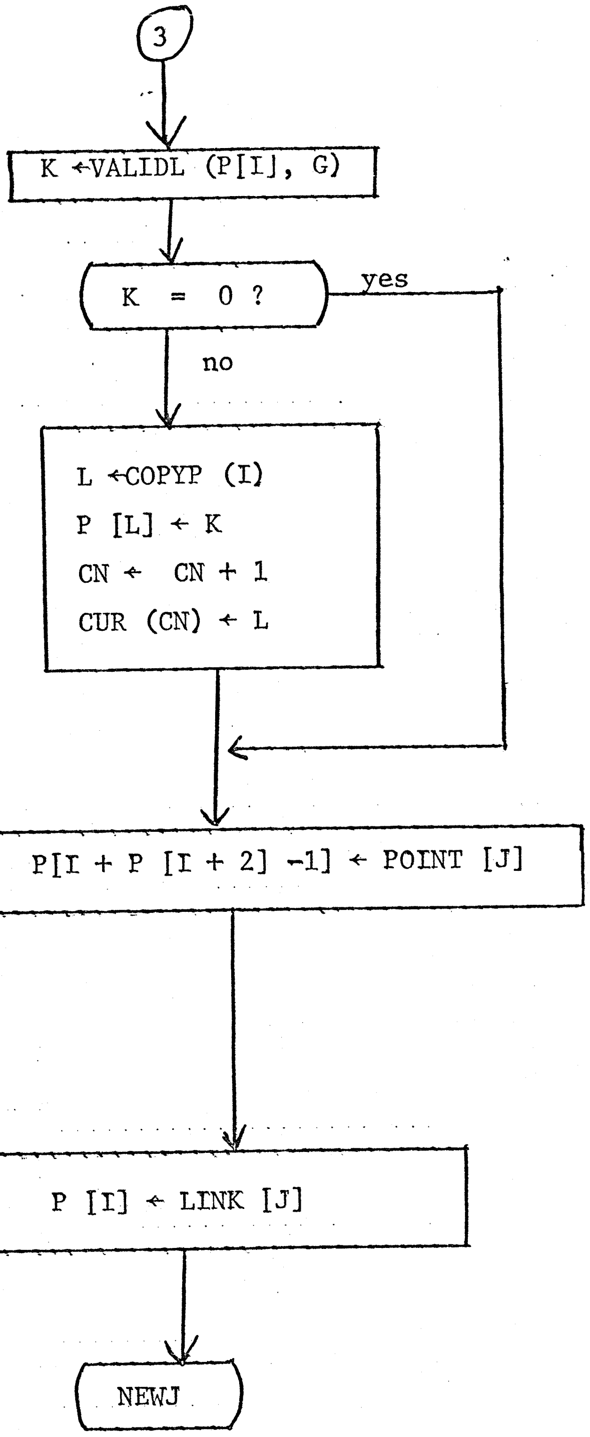


FIGURE 10(7) THE FAST PARSE PROGRAM: PROCESSING AN OUTPUT (BOXED) NODE.

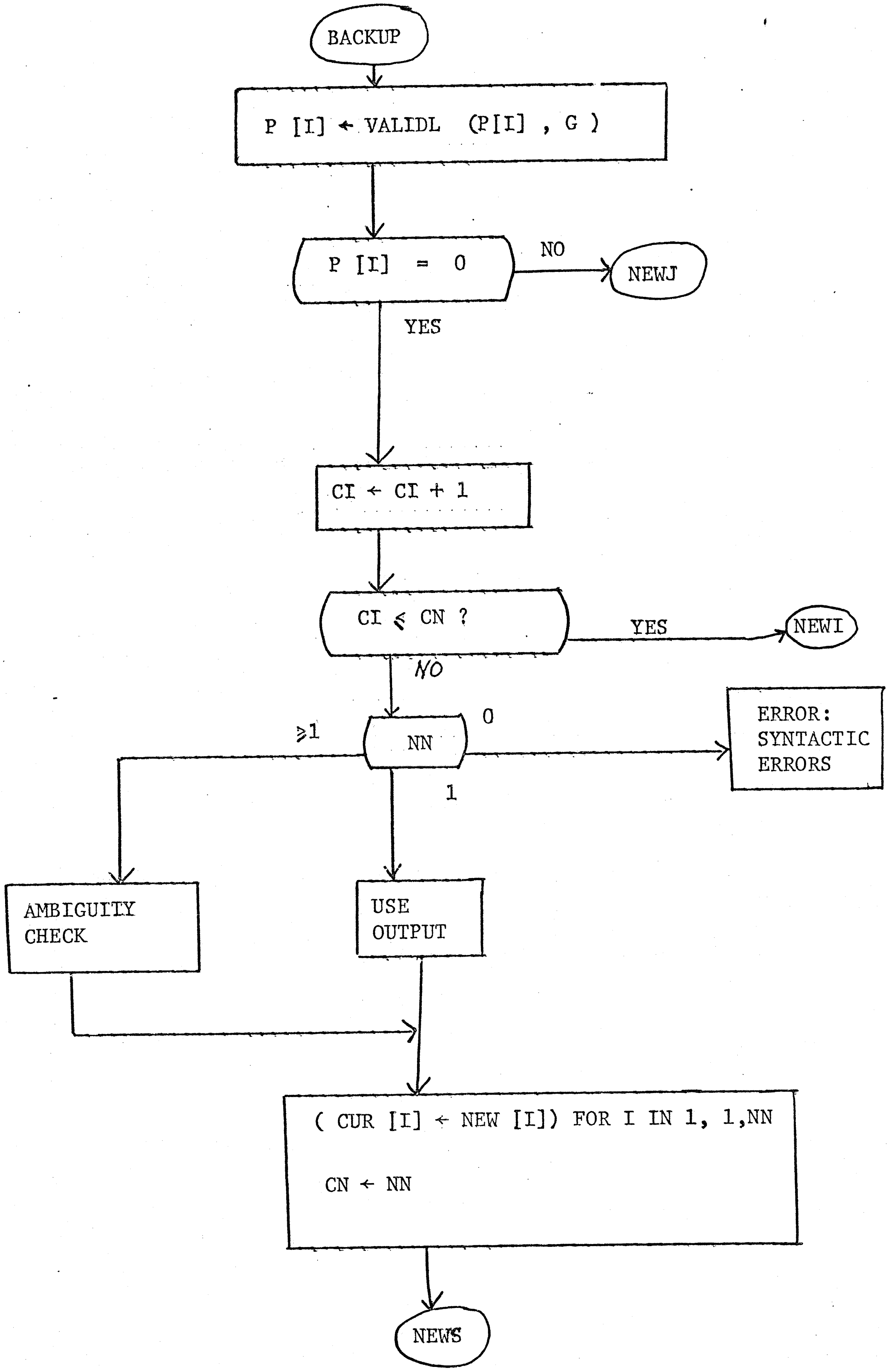


FIGURE 10 (8) THE FAST PARSE PROGRAM: MOVING TO THE NEXT NODE, OR PARSE, OR SYMBOL.

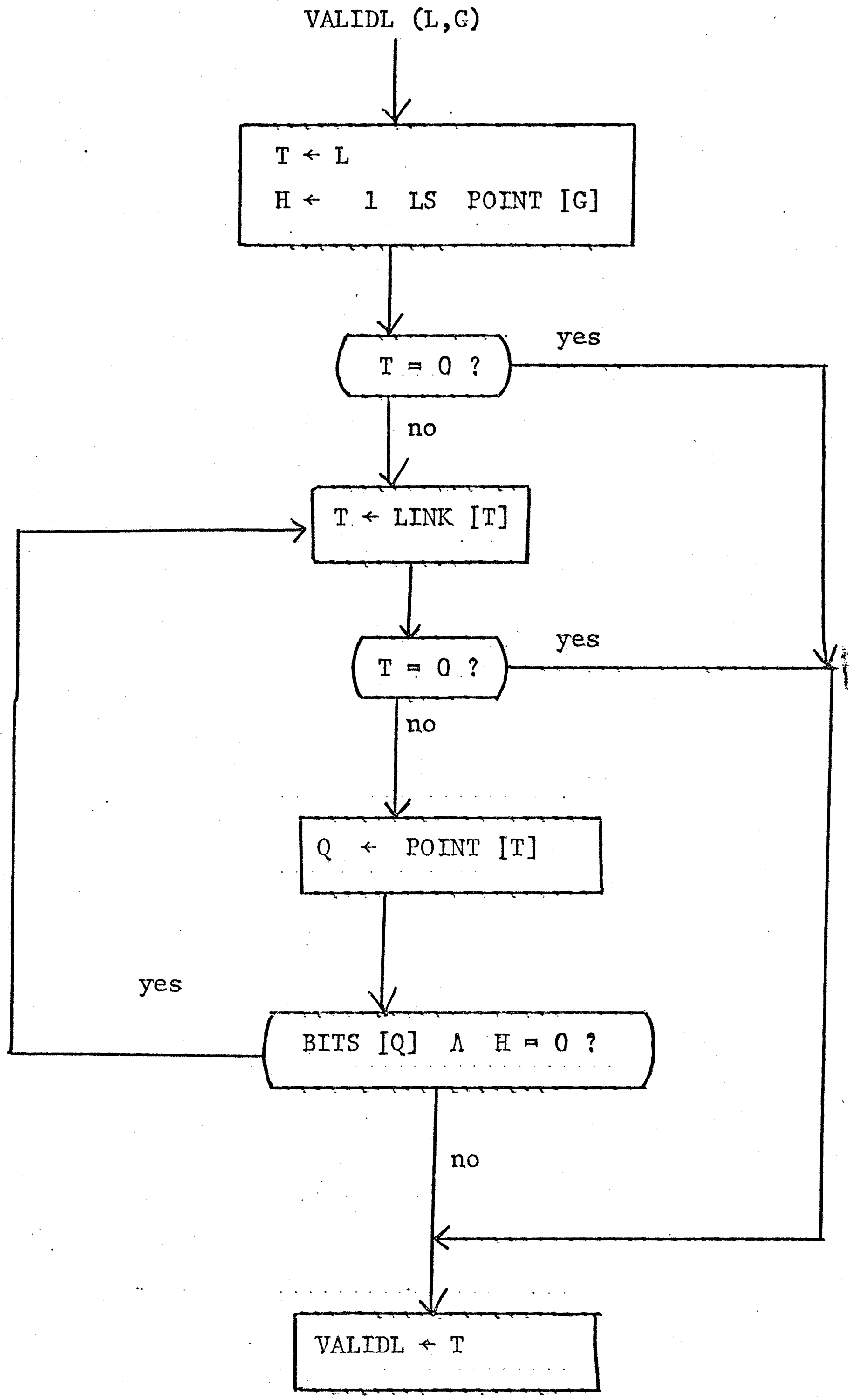


FIGURE 10 (9) THE FAST PARSE PROGRAM: A SUBROUTINE TO FIND THE NEXT NODE WHICH STILL LEADS TO A GOAL G.

```
        SUBR   COPYP  (J)   IS   (

        ( P [H]  =  φ =>   GO TO  GOT )  FOR H IN 1, 30, 301

        ERROR  ( 'OUT OF PARSE SPACE')


GOT:   (P [H + W ]  ←  P [J + W ] )  FOR  W  TO  P [J + 2]

          H )
```

FIGURE 10 (10) THE FAST PARSE PROGRAM: A SUBROUTINE TO FIND SPACE FOR A NEW PARSE, AND COPY AN EXISTING ONE INTO THE NEW SPACE.

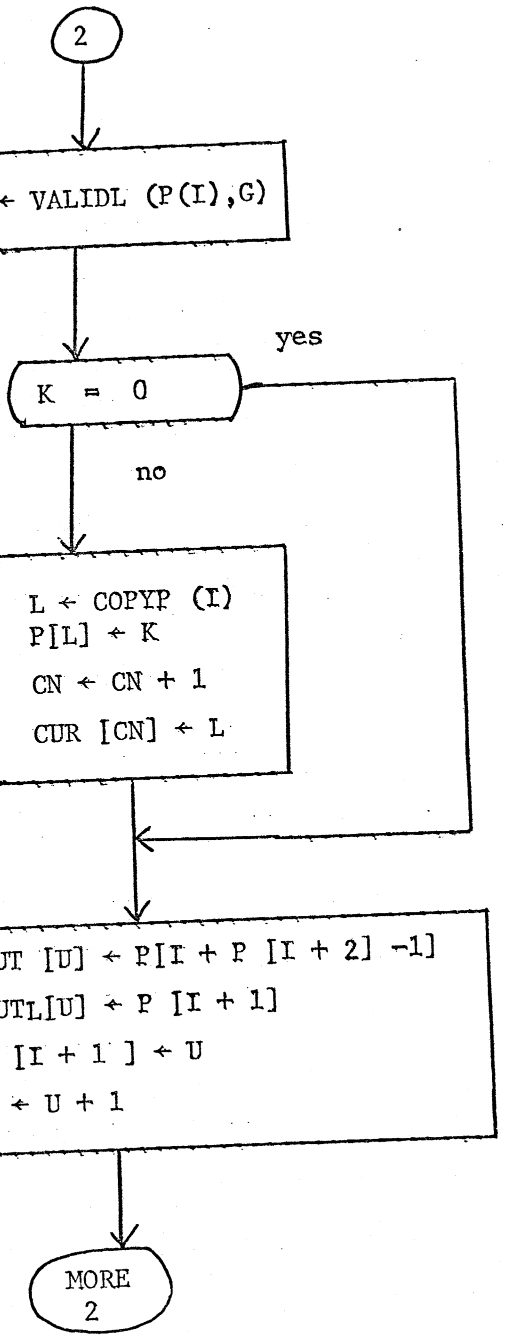


FIGURE 11: CIRCLED NODE PROCESSING FOR THE UNOPTIMIZED BU PROGRAM. FIGURE 10(5-6) REPLACED THIS TO MAKE THE FAST PARSE PROGRAM.

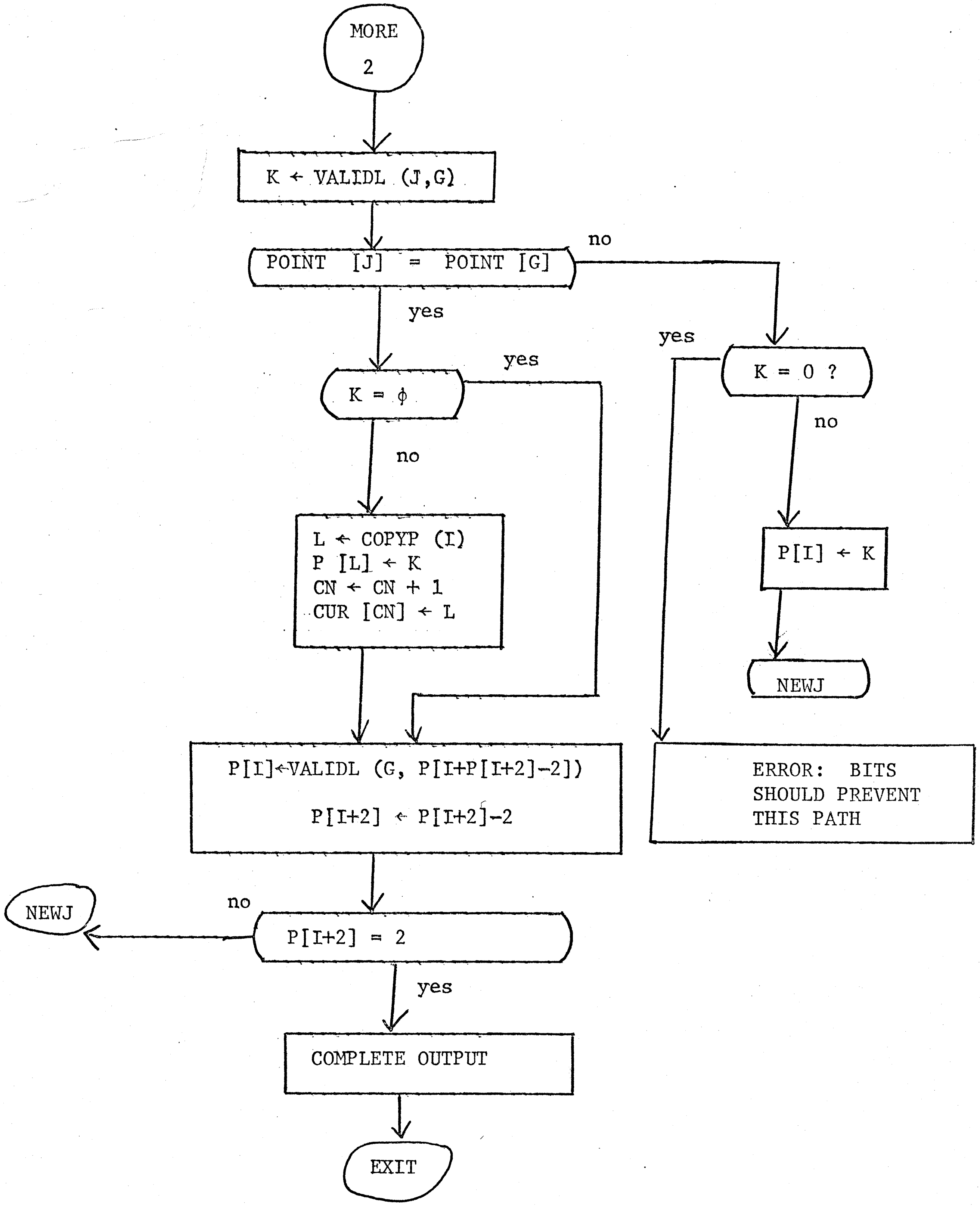


FIGURE 11 (2) CIRCLED NODE PROCESSING FOR THE UNOPTIMIZED BU PROGRAM (CONTINUED)

ARRAY	LENGTH	CONTENTS
P	330	The parses (11 of length 30).
CUR	CN	The locations in P of parses for current symbol.
NEW	NN	The locations in P of parses for the next symbol.
OUT	U	Outputs accumulated until only one parse.
OUTL	U	Links for outputs in OUT.
POINT	M	Value part of a syntax graph node.
LINK	M	Links connecting work for one node.
TYPE	NIX	Type of the node.
BITS	NIX	Indicates circled nodes reachable from this.
DI	DIN	Characters for words produced by lexical analyzer.
ENTRY	DIN	Graph entry points for terminal symbols.
NODE	NXS	Nodes which can come after a circled one for a IS.
NLINK	NXS	Link to next node.
NBITS	NXS	Indicates which circled nodes precede NODE.

FIGURE 12. DATA ORGANIZATION FOR THE FAST PARSE ALGORITHM.